Arrays und Datenstrukturen

Praktikum „C-Programmierung“

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Eugen Betke, Nathanael Hübbe, Michael Kuhn, Jannek Squar

2019-11-04

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Datentypen

`char`	Einzelne Zeichen (1 Byte)
`int`	Integer (üblicherweise 4 Bytes)
`float`	Gleitkommazahl (üblicherweise 4 Bytes)
`double`	Gleitkommazahl (üblicherweise 8 Bytes)
`void`	Unvollständiger Datentyp
`enum`	Aufzählungen (intern Integer)
`struct`	Strukturen
`[]`	Arrays
`*`	Zeiger

sizeof v0

```c
#include <stdio.h>

int main (void) {
    printf("char:   %d\n", sizeof(char));
    printf("int:    %d\n", sizeof(int));
    printf("float:  %d\n", sizeof(float));
    printf("double: %d\n", sizeof(double));
    printf("void:   %d\n", sizeof(void));
    printf("void*:  %d\n", sizeof(void*));
    return 0;
}
```

sizeof v0

```c
#include <stdio.h>

int main (void) {
    printf("char:   %d\n", sizeof(char));
    printf("int:    %d\n", sizeof(int));
    printf("float:  %d\n", sizeof(float));
    printf("double: %d\n", sizeof(double));
    printf("void:   %d\n", sizeof(void));
    printf("void*:  %d\n", sizeof(void*));
    return 0;
}
```

- Mit `sizeof` kann die Größe von Datentypen und Variablen bestimmt werden
  - Erinnerung: Die Größen sind architektur- und implementierungsabhängig
  - Nur die Größe von `char` wird im Standard explizit vorgegeben

Einführung

- C unterstützt Arrays beliebiger Datentypen
  - Intern einfach ein zusammenhängender Speicherbereich
- Auf die Daten wird über einen Index mithilfe von [ ] zugegriffen
  - Dabei finden keine Prüfungen statt
- Die Daten werden zeilenweise im Speicher abgelegt
  - Beispiel: Ein zweidimensionales Array der Größe 2×2

Array v0

```
1  int main (void) {
2      int array[10];
3
4      for (int i = 0; i < 10; i++) {
5          array[i] = i;
6      }
7
8      return 0;
9  }
```

Array v0

```
1  int main (void) {
2      int array[10];
3
4      for (int i = 0; i < 10; i++) {
5          array[i] = i;
6      }
7
8      return 0;
9  }
```

- Arrays werden mit [ ] angegeben
  - [n] steht dabei für ein Array mit n Einträgen
  - Arrays starten mit dem Index 0

Array v1

```
1  void fill_array (int array[]) {
2      for (int i = 0; i < 10; i++) {
3          array[i] = i;
4      }
5  }
6
7  int main (void) {
8      int array[10];
9
10     fill_array(array);
11     return 0;
12 }
```

Array v1

```
1  void fill_array (int array[]) {
2      for (int i = 0; i < 10; i++) {
3          array[i] = i;
4      }
5  }
6
7  int main (void) {
8      int array[10];
9
10     fill_array(array);
11     return 0;
12 }
```

- Arrays können auch als Funktionsparameter übergeben werden

Array v2

```
1  void fill_array (int* array) {
2      for (int i = 0; i < 10; i++) {
3          *(array + i) = i;
4      }
5  }
6
7  int main (void) {
8      int array[10];
9
10     fill_array(array);
11     return 0;
12 }
```

Array v2

```
1  void fill_array (int* array) {
2      for (int i = 0; i < 10; i++) {
3          *(array + i) = i;
4      }
5  }
6
7  int main (void) {
8      int array[10];
9
10     fill_array(array);
11     return 0;
12 }
```

- Intern werden Arrays als Zeiger auf Speicherbereiche behandelt

Array v3

```
1  int main (void) {
2      int array[10];
3
4      for (unsigned int i = 0; i < sizeof(array) / sizeof(*array); i++) {
5          array[i] = i;
6      }
7
8      return 0;
9  }
```

Array v3

```
1  int main (void) {
2      int array[10];
3
4      for (unsigned int i = 0; i < sizeof(array) / sizeof(*array); i++) {
5          array[i] = i;
6      }
7
8      return 0;
9  }
```

- Die Größe eines Arrays kann mit `sizeof` bestimmt werden
  - Allerdings nur an Stellen, an denen der Compiler die Größe kennt

Array v3

```
1  int main (void) {
2      int array[10];
3
4      for (unsigned int i = 0; i < sizeof(array) / sizeof(*array); i++) {
5          array[i] = i;
6      }
7
8      return 0;
9  }
```

- Die Größe eines Arrays kann mit `sizeof` bestimmt werden
  - Allerdings nur an Stellen, an denen der Compiler die Größe kennt
- `sizeof(array)` gibt die Gesamtgröße zurück
  - `sizeof(*array)` die Größe eines Elements

enum v0

```c
#include <stdio.h>

enum {
    ENUM_ZERO,
    ENUM_ONE
};

int main (void) {
    printf("zero: %d\n", ENUM_ZERO);
    printf("one:  %d\n", ENUM_ONE);
    return 0;
}
```

enum v0

```c
#include <stdio.h>

enum {
    ENUM_ZERO,
    ENUM_ONE
};

int main (void) {
    printf("zero: %d\n", ENUM_ZERO);
    printf("one:  %d\n", ENUM_ONE);
    return 0;
}
```

- Mithilfe von `enum` können Integer-Konstanten eingeführt werden
  - Die Nummerierung startet standardmäßig bei `0`

enum v1

```c
#include <stdio.h>

enum {
    ENUM_TWO   = 2,
    ENUM_THREE
};

int main (void) {
    printf("two:   %d\n", ENUM_TWO);
    printf("three: %d\n", ENUM_THREE);
    return 0;
}
```

enum v1

```
1  #include <stdio.h>
2
3  enum {
4      ENUM_TWO   = 2,
5      ENUM_THREE
6  };
7
8  int main (void) {
9      printf("two:   %d\n", ENUM_TWO);
10     printf("three: %d\n", ENUM_THREE);
11     return 0;
12 }
```

- Der Startindex kann angepasst werden
  - Folgende Einträge werden automatisch um eins erhöht

enum v2

```c
#include <stdio.h>

enum {
    BIT_SEVEN = (1 << 6),
    BIT_EIGHT = (1 << 7)
};

int main (void) {
    printf("seven: %d\n", BIT_SEVEN);
    printf("eight: %d\n", BIT_EIGHT);
    return 0;
}
```

enum v2

```c
#include <stdio.h>

enum {
    BIT_SEVEN = (1 << 6),
    BIT_EIGHT = (1 << 7)
};

int main (void) {
    printf("seven: %d\n", BIT_SEVEN);
    printf("eight: %d\n", BIT_EIGHT);
    return 0;
}
```

- Die enum-Einträge können gut für Bit-Werte genutzt werden
  - Auch Bit-Masken etc. sind möglich

struct v0

```
1  struct foo {
2      int bar;
3      char baz;
4  };
5
6  int main (void) {
7      struct foo a;
8      a.bar = 42;
9      a.baz = 'a';
10     return 0;
11 }
```

struct v0

```c
struct foo {
    int bar;
    char baz;
};

int main (void) {
    struct foo a;
    a.bar = 42;
    a.baz = 'a';
    return 0;
}
```

- Ein `struct` ist aus anderen Datentypen zusammengesetzt
  - Auf den Inhalt kann über Variablennamen zugegriffen werden

struct v1

```
1  struct foo {
2      int bar;
3      char baz;
4  };
5
6  int main (void) {
7      struct foo a = { 42, 'a' };
8      struct foo b = { .bar = 42, .baz = 'a' };
9      return 0;
10 }
```

struct v1

```
1  struct foo {
2      int bar;
3      char baz;
4  };
5
6  int main (void) {
7      struct foo a = { 42, 'a' };
8      struct foo b = { .bar = 42, .baz = 'a' };
9      return 0;
10 }
```

- Die Initialisierung ist mit { } möglich
  - Entweder in der richtigen Reihenfolge oder über Namen

struct v2

```
1  #include <stdio.h>
2
3  struct foo {
4      int bar;
5      char baz;
6  };
7
8  int main (void) {
9      printf("sizeof: %lu\n", sizeof(struct foo));
10     return 0;
11 }
```

struct v2

```c
#include <stdio.h>

struct foo {
    int bar;
    char baz;
};

int main (void) {
    printf("sizeof: %lu\n", sizeof(struct foo));
    return 0;
}
```

- Die Größe entspricht nicht immer der Summe der Größe der Komponenten
  - Strukturen werden für effizienten Zugriff mit Padding versehen

struct v3

```c
#include <stdio.h>

struct foo {
    char baz0;
    int bar;
    char baz1;
};

int main (void) {
    printf("sizeof: %lu %lu\n", sizeof(char), sizeof(int));
    printf("sizeof: %lu\n", sizeof(struct foo));
    return 0;
}
```

struct v3

```c
#include <stdio.h>

struct foo {
    char baz0;
    int bar;
    char baz1;
};

int main (void) {
    printf("sizeof: %lu %lu\n", sizeof(char), sizeof(int));
    printf("sizeof: %lu\n", sizeof(struct foo));
    return 0;
}
```

- Die Reihenfolge der Komponenten ist wichtig für das Padding

union v0

```
1  union foo {
2      int bar;
3      char baz;
4  };
5
6  int main (void) {
7      union foo a;
8      a.bar = 42;
9      a.baz = 'a';
10     return 0;
11 }
```

union v0

```
1  union foo {
2      int bar;
3      char baz;
4  };
5
6  int main (void) {
7      union foo a;
8      a.bar = 42;
9      a.baz = 'a';
10     return 0;
11 }
```

- Eine union verhält sich ähnlich wie ein struct
  - Enthält beliebig viele Komponenten, allerdings ist nur eine davon aktiv

union v1

```
1  #include <stdio.h>
2
3  union foo {
4      int bar;
5      char baz;
6  };
7
8  int main (void) {
9      printf("sizeof: %lu\n", sizeof(union foo));
10     return 0;
11 }
```

union v1

```c
#include <stdio.h>

union foo {
    int bar;
    char baz;
};

int main (void) {
    printf("sizeof: %lu\n", sizeof(union foo));
    return 0;
}
```

- Eine union belegt nur so viel Platz wie ihre größte Komponente

Zusammenfassung

- C bietet eine Vielzahl an Möglichkeiten, um eigene Datentypen zu definieren
  - Arrays zur Verwaltung gleicher Daten
  - `enum` zur einfachen Verwaltung von Aufzählungen
  - `struct` zur Definition von zusammengesetzten Strukturen
  - `union` zur Verwaltung zusammengehörender Datenstrukturen